MDD approach for the C programming language

Saravnan.Natarajan, Project Leader, iWave Systems

***Abstract*—Model Driven Development (MDD) addresses the challenges faced by the Embedded system software developers. The main challenge is to meet delivery date in the face of changing requirements, complex system architecture and ever evolving technological platform. However, the MDD supported Object-oriented (OO) modeling uses the full richness of UML for modeling the software intensive systems, including systems-oriented models and real-time and embedded designs. The most common implementation language for real time and embedded systems overall by far is the language C which is functional oriented. In this paper, we examine the possibilities to write object-oriented programs in non-OO language like C. The proposed approach helps to implement an OO design in a small embedded system without a C++ compiler. Introduction to MDD approach, C language implementation of common OO concepts, advantages and disadvantages of MDD method.**

***Index Terms*—C language, Embedded system software, Model Driven Architecture (MDA), MDD, OO concepts, UML**

I. Introduction

MODEL-Driven Development is used to more clearly analyze requirements, define design specifications, test systems concepts using simulation and to automatically generate code for direct deployment on the target hardware[1]. Models and model driven software development are the heart of the MDA approach [2]. Model is a simplified representation used to explain the working of real world system or event [3]. Over all MDA is language, vendor and middleware neutral and therefore a very interesting topic for every software development company [4].

Object-oriented modeling uses the full richness of UML to represent the system and then forward-generates code. This is straight forward, and only tricky aspect of this is the mapping of truly object-oriented futures of UML into C code.

--The first concern is that the C developers' natural instinct is to think about the problem in a functional manner where as most UML and MDD tools force the developer to think and design in an OO environment. Many C developers resist this transition from C environment to an OO environment. In cases where the C developer accepts OO environment, the transition often causes increased ramp-up time for the project.

--The second concern is found in the importance that, external code play in the application. Almost all C development today contains some form of external code. This code may be some special IP that's need to be maintained and reused. The same concern is valid about having to include code generated from a domain specific modeling tool.

--The third concern is the ability to control the generated code. This concern comes up when a project has real-time concerns or if an algorithm needs to be coded in just the right way to maintain fidelity or if underlying hardware needs to be directly accessed for performance.

--The fourth concern is the ability to debug and fine-tune the actual code running on the target hardware and ensure that all the changes or dynamically reflected in the model.

--The fifth concern only applies to C developers targeting highly constrained hardware that requires very efficient code. In this case they need to be assured that the size of the code will not be greater than the code they would generate by hand or else risk an increase in recurring hardware costs.

By keeping the above concern in mind we examine some common idioms that makes transformation of OO concepts to C simple: Classes become structs containing data elements and pointers to "member" functions; generalization is implemented by nesting of the structs representing the base classes; polymorphic operations are generated by producing virtual function tables with name –mangled member functions; and so on [5].

The remainder of this paper is organized as follows: In the next section, we refer to an overview of MDD, the four major steps in MDA-Approach. In section 3, we briefly describe the OO modeling using the UML and their implementation in OO supported language. C language implementation of OO concepts is described in section 4. Finally we conclude the paper in last section.

II. Overview of MDD

A. Overview

Modeling provides a clear understanding of the design intent and eases communication between team members and customers. In addition it makes team based design easier due to a graphical means of portioning the design and defining interfaces. The MDA approach generally separates the system functionality from implementation details. It is a framework for Model-Driven Software Development (MDSD) defined by Object Management Group (OMG).

B. Steps

In the MDA approach there are four major steps:

1. Generation of a Computation Independent Model (CIM)
2. Building a Platform Independent Model (PIM)
3. Transforming the PIM model to Platform Specific Model(PSM)
4. Generate the code out of the PSM model

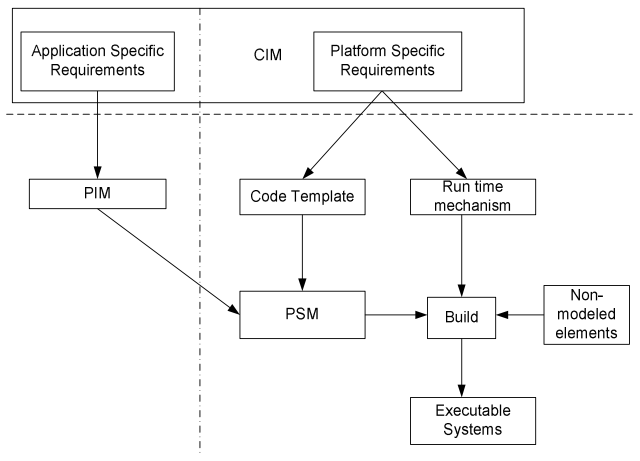

Fig. 1. MDA work flow and key technology elements

Generation of a CIM

The first step is done completely without computer support. This model is common language based and belongs to the Requirements Engineering process. This is further classified as Platform specific and Application Specific process.

Building a PIM model

This model is independent of any implementation technology and has a high level of abstraction. At this level UML is used as modeling language. This step specifies the functionality and behavior. Because the MDA is technology-independent at its core, an application or standard defined in the MDA can implemented equivalently and interoperable on one or several middleware platforms. This model must be extremely detailed: The application will be generated from it and will include only functionality represented explicitly- in the MDA

Transforming PIM to PSM

The complete PIM is stored in Meta Object Facility (MOF) which is input of the mapping step. The map consists of a set of templates and run-time mechanisms. The templates specify the rules for transforming the PIM into executable code.

Generate Code

The run-time mechanisms are a set of implementation utilities required by the generated code. The PSM is integrated with the run-time mechanisms and any non-modeled code including off–the-shelf components or hand-written code to form the executable system.

C. FuncionalC UML profile

A profile is a specialized version of UML that subsets, supersets or extends UML for a particular purpose or vertical market domain. The FunctionalC profile uses a subset of UML for modeling of functionally oriented C-based systems. The primary diagram types defined in this profile are detailed in Table 1.

Diagram type	FC diagram	UML basis diagram	Description
Requirements	Use case diagram	Use case diagram	Represents uses of the system with relations to actors
Structure	Build diagram	Component diagram	Shows the set of artifacts constructed from the source files, such as executables and libraries
	Call graph	Class diagram	Shows the calls and their sequences among sets of functions
	File diagram	Class diagram	Shows the set of .c and .h files and their relations
	Source code diagram	<none>	Shows the generated source code as editable text
Behavior	Message diagram	Sequence diagram	Shows sequences of calls and events sent among a set of files, including passed parameter values
	State diagram	State diagram	Shows the state machine for files and how their included functions and actions are executed as events (whether synchronous or asynchronous) are received
	Flowchart	Activity diagram	Details the flow of control for a function or use case

Table. 1. FuncionalC profile diagrams.

In the next section we will discuss the C language implementation of OO concepts.

III. OO Modeling Using UML

A. Class related concepts

From a UML point of view, a class is the description of a set of objects sharing the same attributes, operations, relations and semantic.

An **attribute** is a named property of class that describes a set of values which can be taken by this property's instances- a class can have zero, one or more attribute(s).

An **operation** is the implementation of a service every objects of the same class can be requested, with the aim of triggering a behavior-in other words, an operations in an abstraction of what can be realized by an object and any objects of the same class.

A **relation** is semantic connection between elements.

A class is symbolized in the UML by a rectangle.

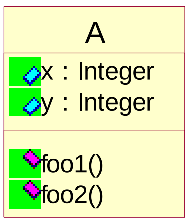

Figure 2 : Class example

From a syntactical point of view, the classical definition from a class found in most Object Oriented language is a set of data and functions/procedure packed all together.

Example 1

```
// C++ syntax
// Definition of class X
class X
{
  // Attributes
  int x;
  int y;

 // Operations
 void foo1();
 void foo2();
};
// End of definition
```

Encapsulation

In UML we have three levels of protection for the attributes and operations: public, protected and private.

Public: available everywhere.
The UML symbol for public visibility is + or nothing.

Protected: accessible only by the class and its subclasses.
The UML symbol for protected visibility is #

Private: accessible only from within the class
The UML symbol for private visibility is -.

B. Relations related concepts

From a UML point of view, a relation is a link between elements. In Object oriented model design, the most important relations are dependency, generalization (inheritance) and association. Here we are going to discuss only about the generalization relationship.

Inheritance

Inheritance is the ability to define new classes and behavior based on existing classes to obtain code re-use and code organization. In other mean it is a mechanism by which new and more specialized classes can be defined in terms of existing class. When a child class (subclass) inherits from a parent class (super class), the subclass then includes the definitions of all the attributes and methods that the super class defines. Objects that are instances of the subclass contain all the data defined by the subclass and its super classes, and they are able to perform all operations defined by this subclass and its super class.

Inheritance is symbolized in the UML as

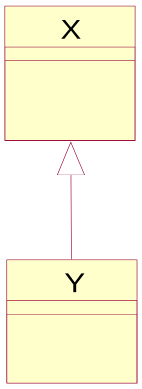

Figure 3 : Single inheritance example

Example 2

```
// C++ syntax
// Single inheritance
class X {
      // Attributes
      int x;
      public :
      /*member functions*/
      } ;
Class Y : public X {
      int y;
      public :
      /* member functions*/
      } ;
```

IV. C LANGUAGE IMPLEMENTATION OF OO CONCEPTS

This section provides the proposal of using OO concepts in C language.

A. Class related concepts in C

In ANSI-ISO C, there is no possibility to package data and operations in such a way as C++ or Java, this is the first problem we have to solve.

The very first idea to promote here is the strict division between definitions of data and operations embedded in a class, linking them with strict naming conventions: From a UML point of view, a class is the description of a set of objects sharing the same attributes, operations, relations and semantic.

<member operation name> = <class name>##<operation name> (## indicating a concatenation)

Basically, data and operations from Example 1 class will be defined this way:

Example 3

```
/* C syntax */
/* From « Xdef.h »
/* Definitions for class A */
typedef struct A_data
{
  // Attributes
 int x;
 int y;
}X_data;
void Xfoo1(X * this,…) ;
/* <Class name>##<operation name> */
void Xfoo2(X * this,…) ;
// End of definition
```

Encapsulation in C

Public: should be defined within data structure in a regular way (de-fault in C).
Protected: use single underscore prefix for protected attributes [6] or pack attributes in a separate structure with a special naming convention.
Private: use double underscore prefix for private attributes or pack attributes in a separate structure with a special naming convention or declare them static in .c file.

B. Relations related concepts in C

Divide the inheritance process in three steps

1. Concatenation of the base and derived class virtual functions (parent class operations at first address as a member named super).
2. Concatenation of the base and derived classes data (parent class data as first address as a member named super).
3. Construction of the derived class itself as usual class.

The proposed C syntax for Example 2 will be defined this way.

Example 4

```
/* C syntax */
/* From « Xdef.h »
/* Definitions for class X */
/* X class data definition */
typedef struct X_data
{
/* public attributes */
type_id attribute_name;
/* protected attributes*/
type_id _attribute_name;
/* private attributes */
type_id __attribute_name;
}X_data ;
/* X class descriptor */
typedef struct X_descriptor
{
/*Pointer to vitual operation foo1*/
ret_type(*foo1)(void* this,..) ;
/*Pointer to vitual operation foo2*/
ret_type(*foo2)(void* this,..) ;
}X_descriptor ;
/*X class definition*/
typedef struct X
{
/* Virtual pointer at first address*/
X_descriptor * vptr ;
```

```
/* C syntax */
/* From « Ydef.h »
/* Definitions for class Y */
/* Y class data definition */
typedef struct Y_data
{
/*super class data at First address*/
X_data super ;
/* public attributes */
type_id attribute_name;
/* protected attributes*/
type_id _attribute_name;
/* private attributes */
type_id __attribute_name;
}Y_data ;
/* Y class descriptor */
typedef struct Y_descriptor
{
/*super class operations*/
X_descriptor super ;
/*Y operation- foo3*/
ret_type(*foo3)(void* this,..) ;
}Y_descriptor ;
/*Y class definition*/
typedef struct Y
{
/* Virtual pointer at first address*/
Y_descriptor * vptr ;
```

/*The embedded data*/ X_data data ; }X ;	/*The embedded data*/ Y_data data ; }Y ;

V.Conclusion

Programming takes discipline. Good programming takes a lot of discipline, a large number of principles, and standards, defensive way of doing things right. Programmers use tools. Good programmers make tools to dispose of routine tasks once for all [7].

The proposals mentioned here will provide a new way of thinking to C developers over the OO concepts. The advancements in the tools for UML make the C language conversion simpler.

References

[1] Jerome L. Krasner, “UML for C developers”, April-2005, www.embeddedforecast.com

[2] Alan Brown, “An Introduction to Model Driven Architecture”, http://www.ibm.com/developerworks/rational/library/3100.html

[3] “Model-Driven Development, Definitions, Challenges, Promises,..”, Artisan White paper

[4] Matthias Gally, “What is the MDD/MDA and where will it lead the software development in the future?”, Department of Informatics at the University of Zurich, Switzerland.

[5] Bruce Powel Douglass, “UML for the C programming language”, June 2009, IBM

[6] Miro Samek, “Portable inheritance and Polymorphism in C”, December 1997, eetindia.com

[7] Axel-Tobias Schreiner, “Object-oriented programming with ANSI-C”, Hollage, October 1993

[8] Carolyn K.Duby, “Accelerating Embedded Software Development with a Model Driven Architecture”, September-2003.